Available online at www.sciencedirect.com

SCIENCE DIRECT®

Journal of Computer and System Sciences 71 (2005) 480–494

JOURNAL OF COMPUTER AND SYSTEM SCIENCES

www.elsevier.com/locate/jcss

An optimal self-stabilizing strarvation-free alternator☆

Mehmet Hakan Karaata

Department of Computer Engineering, Kuwait University, P.O. Box 5969, Safat 13060, Kuwait

Received 24 November 2004; received in revised form 7 May 2005

Available online12 July 2005

Abstract

System refinements from a strong to a weaker model are highly desirable because they allow designers to reason effectively in the strong model. Semantics and fairness refinements are often implemented by alternators preserving the property of stabilization. The existing alternators [Gouda and Haddix, Proceedings of the Third Workshop on Self-Stabilizing Systems (published in association with ICDCS99) The 19th IEEE International conference on Distributed Computing System), IEEE Computer Society Silver Spring, MD, 1999, pp. 48–53; Nesterenko and Arora, Disc99 Distributed Computing 13th International symposium springer, Berlin, 1999, pp. 254–268] provide semantics refinement but fail to implement starvation-freedom. Starvation-freedom requires that no two neighboring processes are scheduled simultaneously; along any interleaved execution, each action is executed infinitely often; no action is infinitely often enabled but never executed; and the number of enabled actions scheduled by the alternator is maximal. In this paper, we first establish the relationships between various mutual exclusion assumptions and the starvation-free alternators (SFAs) under various execution models.Then, as a remedy for the deficiencies of the existing alternators, we propose a fairness refinement for self-stabilizing distributed systems based on a state–space optimal self-stabilizing a (SFA) for arbitrary topologies and verify its correctness.

Keywords: Alternators; Distributed systems; Fairness; Schedulers; Stabilization; Starvation-freedom

1. Introduction

In a distributed system, an execution can be viewed as a sequence of interleaved indivisible actions made by processes. At any instance in time, a number of actions at processes are possible but usually

☆ This research was supported by the Kuwait University Research Administration under grant EO-06/01.
E-mail address: karaata@eng.kuniv.edu.kw.

0022-0000/$ - see front matter © 2005 Published by Elsevier Inc.
doi:10.1016/j.jcss.2005.05.001

only a subset of these are made due to lack of any synchrony. If an action can be executed, we refer to the action as an enabled action, and otherwise as a disabled action. An enabled action can be disabled either by executing the action, or by actions executed by the process itself or the neighboring processes.

A *starvation-free alternator* (SFA) is a process scheduler guaranteeing that no two neighboring processes are scheduled simultaneously; along any concurrent execution, each action is executed infinitely often; no action is infinitely often enabled but never executed; and the number of enabled actions scheduled by the alternator is maximal. The protocols that require starvation-free alternation, also referred to as *strong fairness*, include CSP and other protocols that require handshaking [1]. A more detailed discussion on related issues can be found in [1,11]. A SFA can also be viewed as a system transformer that takes a system with no scheduling assumptions as input and produces a system meeting the above scheduling requirements. Examples of system transformers using alternators are reported in [15,16,25,27]. The transformation in [25] ensures that each concurrent execution of actions is serializable, as defined in the literature of database systems, and so each concurrent execution is equivalent to a serial execution. The transformation in [16] ensures that during any concurrent execution, conflicting actions (those that read and write common variables) are not executed simultaneously. Unfortunately, this transformation is applicable only to systems with linear topologies. Two generalizations of this transformation (that are applicable to systems with arbitrary topologies) are described in [16,17] . The starvation-free generalizations of alternators given in [16,17] are proposed in [20]. Alternators presented in [16,17] simply guarantee that actions (i.e., guarded commands) are executed only if they are not neighboring. In contrast, the transformer proposed in [20] ensures that two actions can be executed simultaneously if the distance between them is two or more instead of more than one as in the case of transformers given in 16]. The transformer presented in [20] uses unbounded variables to implement a SFA. The existing stabilizing alternators are either non-starvation-free or use unbounded counters. In addition, these alternators do not work under interleaved execution.

In this paper, we first establish the relationships between various mutual exclusion assumptions and the SFAs under various execution models. Then, as a remedy of the deficiencies in the existing alternators, we propose a state-space and time optimal *self-stabilizing* SFA for arbitrary topologies and verify its correctness. The proposed transformer is based on the self-stabilizing strong fairness protocol presented in [20]. The strong fairness protocol uses unbounded variables, whereas the proposed protocol uses unbounded variables and is state space optimal. The proposed algorithm, referred to as a SFA, can withstand transient failures and adapt to topology changes in the form of addition and deletion of vertices and edges in a transparent manner. In addition, the SFA concurrently schedules the largest possible number of actions under interleaved execution.

A desirable property of the proposed algorithm is the property of self-stabilization. A self-stabilizing system guarantees that, regardless of the current configuration, the system reaches a legal state in a bounded number of steps and the system state remains legal thereafter. Since the introduction of self-stabilization Dijkstra [8], self-stabilizing algorithms for many fundamental problems in distributed systems have been proposed. For example, self-stabilizing mutual exclusion algorithms for a variety of network classes have been presented [4,6,8]. Self-stabilizing model transformers appear in [20,27]. Self-stabilizing algorithms for a variety of graph theoretic problems are presented in [5,7,18,19,21,23,28]. General techniques for constructing self-stabilizing algorithms are dealt with in [2,24]. Self-Stabilizing algorithms that contain the effects of a single fault are presented in [12–14,17]. Self-stabilizing algorithms are able to withstand *transient failures*. We view a fault that perturbs the state of the system but not the program as a transient

fault. In addition, many self-stabilizing algorithms are capable of gracefully dealing with the dynamic addition and deletion of vertices or edges.

The structure of this paper is outlined as follows: Section 2 motivates the proposed alternator. Section 3 presents the computational model. Section 4 establishes the relationships between mutual exclusion assumptions and the SFAs under the various execution models. Section 5 presents the stabilizing SFA. In Section 6, we show that the algorithm is state–space optimal. Section 7 demonstrates the correctness of the proposed self-stabilizing algorithm. We conclude the paper in Section 8 with concluding remarks and remarks on related issues such as the time optimality.

2. Motivation

In defining the model of a stabilizing system, one needs to choose between the following assumptions concerning the execution of enabled actions.

(i) **Serial execution**: Enabled actions are executed one at a time. This scheduler is defined in [9].
(ii) **Concurrent execution**: Any nonempty subset of enabled actions is executed at a time. All activated actions are executed in a single atomic step simultaneously. That is, they all read their neighbor's states, decide whether to change their state, and move to a new state. This scheduler is defined in [4].
(iii) **Interleaved execution** (power set semantics): Each enabled action is executed atomically at any arbitrary time independently, allowing all possible interleavings of enabled actions.

Serial and concurrent executions can be supported by a *central scheduler*. To support the former semantics, the central scheduler selects an enabled action for execution at a time. Upon completion of the execution, it selects another action, and so on, whereas, to support the latter semantics, the central scheduler selects a set of enabled actions for execution at a time. Upon completion of the execution of these actions, it selects another set of actions for execution, and so on. In interleaved execution, each process independently and atomically starts the execution of an enabled action in the absence of a global deamon such as a central scheduler.

Although, many practical implementations of stabilizing systems require interleaved execution of actions but not serial and concurrent execution, since it is easier to prove stabilization under the assumption of serial and concurrent executions, most stabilizing systems are proven correct under these assumptions. This situation raises an important question. Given that a system is stabilizing under the assumptions of serial and/or concurrent execution, is the system stabilizing under the assumption of interleaved execution? The answer to this question is "no" in general. When a system is transformed to work under the assumption of interleaved execution, additional observable states are introduced leading to problems such as starvation. These problems need to be dealt with by a transformer. For example, consider a system that consists of three processes i, j, k such that process j is a neighbor of both process i and process k. Notice that i and k are processes at distance at most two. Assume that processes i, j, k maintain boolean variables a, b and c, respectively. The actions are defined for the processes i, j, k in Fig. 1.

We refer to the first guard of the algorithm given in Fig. 1 as $G1$ and the corresponding action $A1$, the second guard as $G2$ and the corresponding action $A2$, and so on. Starting in a state where $a = b = true$, $c = false$ and assuming a serial execution of the actions, this system cannot be guaranteed to stabilize to a state where $b = false$ under an unfair scheduler. Starting in a state where $b = true$ and $a \neq c$, if $A1$ and $A3$ are always executed simultaneously and $A1$ always completes before $A3$ under the assumption of interleaved execution, action $A2$ is infinitely often enabled but never executed. Therefore, the system never

{ Program for process i:}
    * [$b \rightarrow a := \neg a;$]
{Program for process j: }
    * [$a = c \wedge b \rightarrow b := \neg b;$]]]
{ Program for process k: }
    * [$b \rightarrow c := \neg c;$]

Fig. 1. Program for processes i, j, k.

converges to a state in which $b = false$. However, if immediately after $A1$'s execution is completed, $A2$ begins to execute after reading the updated value of variable a, then the system stabilizes to a state where $b = false$. In addition, starting from the same initial state since actions $A1$ and $A3$ are always executed simultaneously under the assumption of a maximally concurrent execution, this system will stabilize to a state where $b = true$ and $a \neq c$. Observe that the system stabilizes to a state under the assumption of maximally concurrent execution, however, the same system may or may not converge to a state under the assumption of interleaved execution. Also observe that the state that the system may converge under interleaved execution is not the same as the state the system will converge under the maximally concurrent execution. This negative result is caused by the starvation of process j by the neighboring processes under interleaved execution.

To deal with this negative result we need to identify system transformations that enforce the starvation-freedom property.

3. The model of computation

Following the computational model in [20], we now define our computational model.

A distributed system can be represented by an undirected graph $G = (V, E)$ with vertex (node) set V and edge set E, where $|V| = n$. We assume that each vertex of G is a process with a unique id different from 0. Each process maintains a set of local variables whose values can only be updated by the process itself after inspecting its own local variables and the local variables of its neighbors. Using guarded commands [10], the program of a process i can be expressed as

$$*[G[1] \rightarrow A[1] \Box G[2] \rightarrow A[2] \Box ... \Box G[k] \rightarrow A[k]],$$

where

- each *guard* $G[]$ is a boolean function of the variables of process i and the variables of its neighboring processes;
- each $G[] \rightarrow A[]$ is referred to as a guarded action;
- each *action* $A[]$ reads and updates the variables of process i and reads its neighbors' variables; and
- $*[S]$ corresponds to the repeated execution of the statement S while there exists an enabled guard.
- $\Box$ is called the nondeterminism symbol. One of the guarded actions separated by them is selected nondeterministically in each iteration.

If a guard evaluates to true, we say that the guard, its corresponding action and process are enabled. Note that each action is atomic. Furthermore, to ensure semantic consistency we assume the presence of a distributed deamon that allows non-neighboring processes to execute atomically and concurrently. That is, if two or more guards of the same process or of neighboring processes are enabled at the same

time, then for each neighborhood the *scheduler* arbitrarily selects a single guard among these enabled guards and allows the execution of the corresponding action to be completed atomically. A self-stabilizing algorithm implementing this can be adapted from [16,26]. Therefore, an implementation of this is omitted. In addition, we assume that the scheduler is *weakly fair*, i.e., any action remaining enabled is eventually executed. Note that these assumptions are made on the alternator but not necessarily on the program input to the alternator.

The *state* of a process is composed of the set of variables at the process. The *system state* is the cartesian product of the states of the processes in the system. The system state before the system is started is referred to as the *initial state*.

We refer to the algorithm on which SFA is to be enforced as the *application protocol*. We assume that the application protocol is in the form of guarded commands and executes under interleaved execution. Define the application protocol as a distributed system represented by $G = (V, E)$ where each process has only one action. Let A be the set of actions of the application protocol. For each action $i \in A$ of the application protocol, there exists a node i in G. For two nodes $i, j \in V$, $(i, j) \in E$ iff action $i \in A$ reads a variable written by action j or action $j \in A$ reads a variable written by action $i \in A$. That is, if two actions $i, j \in A$ share a variable v, where i reads v and j writes into v or vice versa, actions i, j are said to be neighboring actions. Define the SFA as a distributed algorithm that has the same topology as that of the application protocol such that the SFA has one process corresponding to each action of the application protocol. In addition, processes i, j of the SFA corresponding to neighboring actions i, j, respectively, of the application protocol are said to be neighboring processes.

Throughout the paper, by action we refer to an action of the application protocol unless it is explicitly stated that it is an action of the (SFA), and by process or node to its corresponding process of the (SFA). In addition, we refer to the action corresponding to process $i \in V$ as action i and vice versa.

4. Properties of stravation-free alternators

In this section, we establish the relationships between various mutual exclusion assumptions and the SFAs under the three execution models.

An algorithm is said to implement a SFA if it satisfies the following safety and liveness properties:

Strong-Fairness (Liveness): No action is infinitely often enabled but never executed.
Mutual Exclusion (Safety): In a locality, i.e., among processes at distance one and two, simultaneous execution of actions of the application protocol is not allowed.
Deadlock-Freedom (Safety): For each state of the SFA, in each locality, if enabled actions of the application protocol exist, exactly one of them is eventually allowed to be executed.
Weak-Fairness (Liveness): An enabled action, if it remains enabled, is eventually executed.

First, we show that starvation-freedom cannot be implemented unless the processes at distance one or two execute their actions mutually exclusively under interleaved execution. Note that the starvation-freedom property is also referred to as *strong fairness* in the literature and is defined as follows. If no action is infinitely often enabled but never executed in a system, then the system is said to be strongly fair.

We define the *locality* for each process i as the set of processes at distance one and two from process i including itself. If executions of multiple actions are allowed in a locality at the same time, starvation-freedom cannot always be guaranteed as stated by the following lemma. Allowing actions to be executed at the same time does not necessarily mean that they will be executed simultaneously. It merely means that the SFA allows them to be executed at the same time; and they can actually overlap in an arbitrary manner.

Lemma 1. *There exists an application protocol A, for which if executions of multiple actions of A at distance one and two are always allowed at the same time by the SFA, starvation-freedom cannot be implemented.*

Proof. The proof is analogous to that of Lemma 2 in [20], and hence, is omitted.

A SFA can skip enabled actions any finite number of times, so it could allow processes at distance two to concurrently execute for finite periods but not for infinite periods as stated by Lemma 1. However, two actions i and j of the application protocol in a locality can be allowed to always execute their actions at the same time if there exists no action k such that k can be enabled by one of the two actions and disabled by the other. Also, observe that an algorithm implementing starvation-freedom needs to examine the guards and actions of the application protocol to determine whether two actions can be allowed to execute at the same time or not. Since we assume that the proposed protocol knows only whether an action is enabled or not but not the nature of guards and actions of the application protocol, each action is required to execute its actions mutually exclusively in its locality.

Thus far, we showed that mutually exclusive action execution in each locality is a requirement for the SFA. Now, we have to show that mutual exclusion in each locality and neighborhood implements the SFA under interleaved and serial/concurrent executions, respectively. The following lemmas establish the relationships between various mutual exclusion algorithms and (SFAs).

Lemma 2. *A mutual exclusion algorithm that executes actions in each locality mutually exclusively while allowing the maximum number of concurrent action executions in the entire system implements the SFA under interleaved execution.*

Proof. Assume that mutual exclusion is implemented in each locality, i.e. when an action is executed no other action the locality of i is executed simultaneously, an action will eventually be executed (deadlock-freedom), and each enabled action will eventually be executed (no lockout).

Clearly, process $i \in V$ cannot be scheduled simultaneously with one of its neighbors by the mutual exclusion assumption. Therefore, we have the first property of the SFAs.

If process i is scheduled only finitely many times for an interleaved execution, then there exists a suffix of the execution in which i is not scheduled at all. Notice that since mutual exclusion is implemented in the locality of i, each time i is enabled, its enabledness status is observed. Clearly, the starvation of i along any suffix of the execution violates the mutual exclusion assumption. Hence, process i is scheduled infinitely many times along any interleaved execution.

Let i be a process infinitely often enabled but never executed. Since, each time i is enabled, its enabledness status is observed, there exists a neighbor j of i such that i and j are both enabled together

infinitely often in the execution which disables i when executed. Clearly, this violates the mutual exclusion assumption. Note that if process i remains enabled but never scheduled for execution, the mutual exclusion is violated. Hence, the proof follows.

Similarly, a mutual exclusion algorithm guaranteeing mutual exclusion in each neighborhood implements the SFA under serial and concurrent executions as stated by the following lemma. Its proof is analogous to that of Lemma 2 and hence, is omitted.

Lemma 3. *A mutual exclusion algorithm that executes actions in each neighborhood mutually exclusively while allowing the maximum number of concurrent action executions in the entire system implements the SFA under serial and concurrent executions.*

5. Algorithm

In this section, we present an improved version of the algorithm given in [20] for implementing the (SFA) that allows concurrent execution of the actions of the application protocol.

Now, we briefly describe the strategy employed by the SFA. The alternator ensures that the least recently executed action of the application protocol is never disabled by neighboring actions and is executed before any neighboring action is executed. Since the alternator guarantees that each enabled process eventually becomes the least recently executed process in its locality and executes its action, it implements starvation-freedom. Also, the SFA allows the execution of a single action of the application protocol at any given time in a locality which is shown to be a requirement for starvation-freedom by Lemma 1. After action i is executed, the SFA reexamines the enabledness status of the actions of the application protocol and updates the priorities in the locality. Consequently, it allows another action to be executed in the locality. In addition, to obtain the highest degree of concurrency, the proposed algorithm exploits the facts that most computer networks are sparsely connected and non neighboring actions cannot disable each other.

We now describe the implementation details of the algorithm. In a locality, simultaneous execution of actions of the application protocol is not allowed since such actions may violate starvation-freedom as discussed earlier. The proposed algorithm implements mutual exclusion in a locality as follows: we have a single pointer associated with each process $i \in V$ pointing to itself or one of its enabled neighbors with the highest priority among neighbors of i. Each process $i \in V$ of the SFA executes its corresponding action of the application protocol only when it and all its neighbors point to process i. Since when process i is executing an action of the application protocol, all its neighbors point to process i and no neighbor can execute an action to change its pointer by our mutual exclusion assumption among neighbors, no process at distance one or two from process i can execute its action of the application protocol at the same time.

In addition, each process maintains a priority relative to its neighbors' priorities. The proposed algorithm ensures that the priority of a process changes when it executes an application protocol action: it gets a new priority lower than any priority in its neighborhood. Furthermore, it guarantees that only the enabled action corresponding to the process with the highest priority among enabled actions in its locality is executed. As a result, a process not executing an action eventually obtains the highest priority among enabled actions in its locality, and when enabled, executes its action. We assume that each vertex i has a unique $id\ i$.

The proposed algorithm distinguishes itself from the strong fairness protocol presented in [20] in the scheme employed for the maintenance of the priorities. The strong fairness protocol maintains an unbounded counter per process. In a locality, the process with the least counter is always allowed to execute its corresponding action of the application protocol. Upon completion of the action execution, the process assigns a counter value larger than the largest counter in its locality. Clearly, the counters are incremented indefinitely.

Now, we describe the way in which the relative priorities of processes are maintained using only bounded variables by the proposed algorithm. We associate a direction with each edge $(i, j) \in E$ to determine the relative priorities of process i and its neighbor j. Each process $i \in V$, for each neighbor j of i, maintains a single-digit binary variable $pri(i, j)$. The direction of priority edge (i, j) is determined by two variables $pri(i, j)$ and $pri(j, i)$ maintained by processes i and j, respectively, and the unique process id's of processes i and j. The priority between two processes i and j is defined as $j <_p i$ iff $\{pri(i, j), pri(j, i)\} \in \{\{0, 1\}, \{1, 0\}\} \wedge i < j)$ or $\{pri(i, j), pri(j, i)\} \in \{\{0, 0\}, \{1, 1\}\} \wedge j < i)$. If $j <_p i$, then process i is said to have a higher priority than process j. This scheme of assigning directions allows the direction to be changed by changing one of the priority values maintained by processes incident on the priority edge. For instance, upon taking its corresponding action, it updates its priority with respect to its neighboring processes, by making all priority edges incident on it outgoing indicating that it has the least priority.

To facilitate the description of the algorithm, we introduce several internal functions (variables) that return the current state of the process:

$pri : V \text{x} V \rightarrow \{0, 1\}$ denotes the direction of the priority edge (i, j) with the help of another priority value as explained earlier. The direction of the edge determines the relative priorities of its endpoints.

$f : V \rightarrow \{true, false\}$ denotes the recognition of the enabledness status of action i by process i. If $f(i)$ is true, then action i is enabled, process i has recognized the enabledness of action i and requesting a permission to execute action i. Otherwise, action i is disabled or it is enabled but process i is not yet requesting a permission to execute action i. We refer to $f(i)$ as the f-value of process i.

$m : V \rightarrow \{0, \cdots, n\} \cup \{\Phi\}$ denotes the process id of the neighboring process with the highest priority among the neighboring processes whose f-values are true, or its own id if it is enabled and has the highest priority among the neighboring processes whose f-values are true. If no such process exists, then it denotes Φ which is not a valid process id. We refer to $m(i)$ as the m-value of process i.

$c : V \rightarrow \{0, \ldots, n\}$ denotes whether or not the process is in a cycle. If the c-value of a process is C or larger, where C is a large constant, then the process is in a cycle. We refer to $c(i)$ as the c-value of process i.

We also use the following notation: $action(i)$ denotes the action of the application protocol associated with process i of the SFA. $e.i$ indicates the enabledness status of $action(i)$, that is, if $e.i$ is true, then the action is enabled. otherwise, it is disabled. In addition, N_i denotes the set of neighboring processes of i and N_i^+ called the *neighborhood* of process i, denotes the set of neighboring processes of i and i itself.

Now, before we present the algorithm, we need the following notation.

$low\_pri(i)$ ensures that process i receives the lowest priority among its neighbors.

$$
\begin{array}{l}
\{\ \text{Starvation-free alternator for process } i\ \} \\
*\big[m(i) \neq mf.i \quad \longrightarrow \quad m(i) := mf.i; \\
\ \square\ \neg e.i \wedge f(i) \quad \longrightarrow \quad f(i) := false; \\
\ \square\ e.i \wedge \neg f(i) \wedge \forall_{j \in N_i}(m(j) \neq i) \quad \longrightarrow \quad f(i) := true; \\
\ \square\ \forall_{j \in N_i^+}(m(j) = i) \wedge (e.i \wedge f(i)) \quad \longrightarrow \quad action(i); f(i) := false; low\_pri(i); \big]
\end{array}
$$

Fig. 2. The SFA (Algorithm SFA).

We need to define the following function used in the definition of function $low\_pri(i)$.

$$
ch\_pri(i)\text{denotes} \begin{cases} 0 \text{ if } pri(j,i) = 0 \wedge i < j, \\ 1 \text{ if } pri(j,i) = 1 \wedge i < j, \\ 0 \text{ if } pri(j,i) = 1 \wedge j < i, \\ 1 \text{ if } pri(j,i) = 0 \wedge j < i, \end{cases}
$$

The function $low\_pri(i)$ is implemented as follows:

do for each $j \in N.i$
$pri(i, j) := ch\_pri(pri(j, i))$;
$c(i) := 0$;

In addition, we will use the following notation: $h.i$ denotes the process with the highest priority among processes in N_i^+ and whose f-values are true. Furthermore, function $mf.$ is defined as follows.

$$
mf.i \text{ denotes} \begin{cases} i, & e(i) \wedge (i = h.i), \\ j, & \neg e(i) \wedge \exists_{j \in N_i}(j = h.i), \\ \Phi, & \text{otherwise}, \end{cases}
$$

The algorithm called *SFA* is given in Fig.2 implementing the above described strategy. Throughout the paper, we refer to the first guard of the SFA as $G1$ and the corresponding action $A1$, the second guard as $G2$ and the corresponding action $A2$, and so on. Note that we assume that each process starts execution with arbitrary initial c-values, m-values, pri-values and f-values.

Now, we briefly describe each guarded statement of the algorithm. The first guard $G1$ and action $A1$ ensure that for each process $i \in V$, $m(i)$ points to an enabled neighbor (or itself) with the highest priority among the neighbors of process i. $G2$ and $A2$ are used to lower the flag of each process $i \in V$ by assigning false to $f(i)$ if action i is disabled. Note that a process raises its flag to indicate a request to execute an action of the application protocol and lowers its flag to indicate that it has no desire to execute the action. Observe that the meaning of the term $\forall_{j \in N_i}(m(j) \neq i)$ in $G3$ plays an important role in the algorithm. The term indicates that every neighbor of i recognizes that i has executed its action. $G3$, $A3$ pair guarantee that if action i is enabled but its flag is down ($f(i) = false$) and every neighbor of i recognizes that i has executed its action, i.e., $\forall_{j \in N_i}(m(j) \neq i)$, then process i raises its flag by assigning true to $f(i)$. Guard $G4$ and action $A4$ are responsible for the following. When each process $j \in N_i$ agrees that process i is the next to execute its action by setting its m-value $m(j)$ to i, i is enabled, and its flag $f(i)$ is true, then the corresponding action $action(i)$ is executed. In addition, $f(i)$ is assigned false indicating that process i is not competing with its neighbors to execute its corresponding action. Also, process i is made the

$$
\begin{array}{l}
\{\ \text{The cycle breaking algorithm for process } i\ \} \\
*\Big[\neg\exists_{j\in N_i}(i <_p j) \wedge (c(i) \neq 0) \quad \longrightarrow \quad c(i) := 0; \\
\ \Box\ \exists_{j\in N_i}\forall_{k\in N_i \wedge i<_p k}(i <_p j \wedge c(j) \geq c(k)) \wedge c(j) \leq C \wedge c(i) \neq c(j)+1 \\
\qquad\qquad\qquad\qquad\qquad\qquad \longrightarrow \quad c(i) := c(j)+1; \\
\ \Box\ (c(i) > C) \qquad\qquad\qquad\qquad \longrightarrow \quad low\_pri(i);\Big]
\end{array}
$$

Fig. 3. The cycle breaking algorithm.

process with the lowest priority in its neighborhood by executing $low\_pri(i)$ which assigns the lowest priority among its neighbors to i.

Consider a directed graph $P = (V, A)$, referred to as the priority graph such that $(i, j) \in A$ iff $(i, j) \in E$ and $i <_p j$. Algorithm SFA works correctly if the initial priority graph is acyclic; however, if P is cyclic the processes may be deadlocked. We use the following mechanism to break the cycles in the priority graph. Each process i maintains a counter $c(i)$ referred to as c-value of process i. $c(i)$ is assigned 0 if i has no successor. Otherwise, $c(i)$ is assigned the maximum c-value belonging to a successor of i plus one. If i is contained in a cycle, $c(i)$ eventually exceeds $C \leq n$ by the weak fairness assumption. Then, after $c(i) > C$ holds, i is assigned the lowest priority in its neighborhood breaking the cycle. It is easy to see that even if i is assigned the lowest priority in its neighborhood when $c(i)$ exceeds a small constant such as 5, the cycles can be eliminated while maintaining the starvation-freedom property of the alternator. As a result, the SFA can be implemented only through local computations involving only those processes that are at distance 5 from each other. Note that C can be chosen as small as 3, which is the size of the smallest cycle in the graph. Each cycle in the priority graph is eventually destroyed and the priority graph eventually becomes a DAG. After the priority graph becomes a DAG, it remains as a DAG under normal system execution. The algorithm implementing the above mechanism is given in Fig. 3 .

6. State–space optimality

Define a total ordering of neighboring processes $<_t$ such that $p_k <_t p_l$, where k, l are neighboring iff after the last move by node l, node k has made a move. We now show that neighboring processes in a system must be totally ordered in order to implement the SFA.

Lemma 4. *Let $E = E_0E_1 \cdots E_k \cdots$ be a strongly fair execution of an arbitrary protocol PR. Also, let protocol SF be the protocol implementing strong fairness for protocol PR. There exists a suffix of execution E' in which a total ordering is always maintained for each pair of neighboring processes.*

Proof. (By contradiction). Assume the contrary. Then, we know that for two arbitrary processes $i, j \in P$, $p_i \not<_t p_j$ and $p_j \not<_t p_i$ hold infinitely often. Let protocol PR be such that only processes i and $j \in P$ are enabled only when for processes $i, j \in P$, $p_i \not<_t p_j$ and $p_j \not<_t p_i$ hold. Otherwise, processes i and j are disabled. Also, assume that when one of i or j is executed, the other one is disabled.

Since the protocol SF is a deterministic one and processes i and j are unrelated, it always selects the same process between i and j when both are enabled. Without loss of generality, we say that p_i is executed each time when both p_i and p_j are enabled. Now, process p_j is enabled and disabled infinitely

often without making a move which contradicts our initial assumption that execution E is starvation-free. Hence, the proof follows.

The following proposition shows the number of states and bits required to maintain the total ordering of neighboring processes. The proof is trivial, and hence, is omitted.

Proposition 5. *At least $|E|$states and $|E|$ bits are required to maintain the total ordering of all the pairs of neighboring processes.*

The following lemma shows that algorithm SFA is state–space optimal. The lemma follows from Proposition 5 and the fact that the proposed algorithm uses $O(|E|)$ bits in total.

Lemma 6. *Algorithm SFA is state–space optimal.*

7. Correctness

In this section, we demonstrate the correctness of our algorithm.

An algorithm implementing starvation-freedom is said to be *self-stabilizing* if it satisfies the following liveness properties starting from any arbitrary state:

- No action is infinitely often enabled but never executed.
- An enabled action is eventually executed.

We now show that Algorithm SFA is self-stabilizing by establishing these properties.

We first establish that the priority graph P' eventually becomes a DAG and remains a DAG thereafter by the following two lemmas.

Lemma 7. *Priority graph P eventually becomes a DAG.*

Proof. Let i be a process in a cycle of priority graph P. Observe that process j's c-value eventually exceeds C by the weak fairness assumption (see the cycle–breaking algorithm). Again by the weak fairness assumption, each such process i eventually executes $low\_pri(i)$ and breaks the cycle (see $A3$ of the cycle–breaking algorithm). Hence, the proof follows.

Lemma 8. *If priority graph P is a DAG, P remains a DAG under normal system execution.*

Proof. First observe that priority graph P is altered only by executing the last action ($A4$) of algorithm SFA. After $A4$ or $low\_pri(i)$ is executed by process i, all the arcs incident on i become outgoing. It is easy to see that this action causes neither i nor a neighbor of i to be included in a cycle. Hence, the proof follows. □

We now establish the deadlock-freedom of the algorithm by the following lemma. For this purpose, we need to show that at any instance in time an enabled action of the application protocol, if any, is allowed to be executed in each locality.

The proofs of Lemmas 9, 12, and 13 are analogous to those of Lemmas 1, 5, and 6 in [20], respectively and, hence, are omitted.

Lemma 9. (*Deadlock Freedom) For each state of the SFA, in each locality, if enabled actions of the application protocol exist, exactly one of them is eventually allowed to be executed.*

Now, we describe the way in which the SFA works in more detail. The algorithm guarantees that actions execute mutually exclusively in a locality as follows. In order for a process i to execute an action of the application protocol, all its neighbors and itself must have set their m-values to i (see $G4$). Since if all the neighbors of process i have set their m-values to i, no other action in the locality of action i can execute concurrently with action i. This is stated in the following proposition.

Proposition 10. *(Mutual exclusion) The SFA guarantees that for each process i, while process i is executing the action of the application protocol, no other process in the locality of process i can execute an action of the application protocol.*

Guaranteeing that no two actions of the application protocol are allowed to be executed at the same time is not sufficient for implementing starvation-freedom. It should also be guaranteed that no action is infinitely often enabled but never executed. In addition, an action remaining enabled must eventually be executed. Note that the latter is guaranteed by our weak fairness assumption. To guarantee the former we employ a priority–based mechanism where the process corresponding to an action that is enabled and disabled infinitely often or remains enabled eventually obtains the highest priority in its locality. We need to show that our algorithm ensures that an action that is enabled and disabled infinitely often or remains enabled eventually obtains the highest priority and executes.

The causes of the transitions can be as follows:

- An execution of an action of the application protocol or the SFA by i or j. (Notice that an action by one may disable the other. Also notice that an action may change f-values and/or e-values of these processes.)
- An execution of an action of the application protocol by a neighbor of i or j.
- An execution of an action of the application protocol by a common neighbor of i and j.

Lemma 11. *Let i and j be two neighboring processes such that $j <_p i$. Action i can be disabled by action j at most once during any time interval in which $j <_p i$ holds.*

Proof. We now show that actions (transitions) by action j disabling action i cannot be repeated more than once during any time interval in which $j <_p i$ holds.

Observe that if $e.i \wedge f(i) \wedge e.j \wedge f(j)$ and $mf.i = j$ holds due to arbitrary initialization, action j can be taken as disabling action i. Also observe that $mf.i$ cannot again become equal to j when $j <_p i$ and $e.i$ holds. Now, consider the state after action i is disabled by action j. Notice that in this state, $j <_p i$ holds. In order for action i to be disabled by action j for the second time, action i has to be enabled by action j or another neighbor of i. If action i is enabled by a move of action j, since $mf.i$ cannot again become equal to j when $j <_p i$ and $e.i$ hold, action j cannot disable action i. Hence, for action i to

be disabled by a move by action j, action i must be enabled by an action of a neighbor k of i such that $k \neq j$. Notice that prior to taking action k, $mf.i = k$ has to hold. Now, immediately after taking action k, we have either $e.i \wedge \neg f(i) \wedge e.j \wedge f(j)$ and $mf.i = k$, or $e.i \wedge f(i) \wedge e.j \wedge f(j)$ and $mf.i = k$. Since $mf.i$ cannot again become equal to j when $e.i$ holds, action j cannot disable action i for the second time. Hence, the proof follows.

Lemma 12. *Let i and j be two neighboring processes. If process i does not execute, process i can be disabled by process j at most twice.*

Note that since action i cannot be disabled infinitely often by the above lemma, eventually $e.i$ remains true after it is enabled. The following lemma shows that an action that remains enabled eventually executes.

Lemma 13. (Liveness). *If $e.i$ remains true forever, then action i eventually executes.*

The following lemma immediately follows from the above Lemmas 9, 1, 12, 13 and Proposition 10.

Lemma 14. (Correctness). *Algorithm SFA is stabilizing and implements starvation-freedom.*

8. Conclusions

In this paper, we first established the relationships between various mutual exclusion assumptions and the SFAs under various execution models. We then presented a state–space optimal SFA. The alternator deals with a generalized version of the dining philosophers problem. The proposed alternator implements mutual exclusion among processes at distance two or less, whereas a solution to the dining philosophers problem implements mutual exclusion among neighboring processes. It is easy to see that the locality of mutual exclusion is important and determines whether the problem at hand is the general mutual exclusion, the dining philosophers, or the SFA (strong fairness) problem. When mutual exclusion is implemented in the locality of the entire network, among processes at distance at most two, and among neighboring processes, we obtain solutions to the general mutual exclusion, the SFA and the dining philosophers problems, respectively.

In [20], it is shown that the strong fairness algorithm is a highly concurrent scheduler for strong fairness. In addition, an upper bound on the number of actions made by the strong fairness protocol for an action of the application protocol is given. These also hold for the starvation-free scheduler. The proofs are analogous, and hence, are omitted.

We showed that the SFA executes $O(d)$ actions for each action of the application protocol, where d is the maximum degree in the graph. We conjecture that no algorithm can implement starvation-freedom with less than $O(d)$ actions of the SFA per action of the application protocol. That is, our algorithm is time optimal. It can be argued that since process i needs to execute its action mutually exclusively in its locality, each neighbor of i at distance two needs to know that i is to execute its action of the application protocol. Note that this can only be guaranteed through actions of neighbors of process i. If each neighbor of process i does not execute an action of the SFA, it cannot be guaranteed that no neighbor at distance two executes its actions. Since i can have at most d neighbors, the optimal algorithm executes $O(d)$ actions.

Acknowledgments

We thank Ted Herman and the anonymous referees for their suggestions and constructive comments on the manuscript. Their suggestions have greatly enhanced the paper.

References

[1] K.R. Apt, N. Francez, S. Katz, Apprasing fairness in languages for distributed programming, Distributed Computing 2 (1988) 226–241.
[2] A. Arora, M.G. Gouda, Distributed reset, IEEE Trans. on Comput. 43 (1994) 1026–1038.
[3] B. Awerbuch, B. Patt-Shamir, G Varghese Self-stabilization by local checking and correction, In: FOCS91 Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science, (1991) pp. 268–277.
[4] G.M. Brown, M.G. Gouda, C.L. Wu, Token systems that self-stabilize, IEEE Trans. Comp. 38 (6) (1989) 844–852.
[5] S.C. Bruell, S. Ghosh, M.H. Karaata, S.V. Pemmaraju, Self-stabilizing algorithms for finding centers and medians of trees, SIAM Journal on Comput 29 (2) (1999) 600–614.
[6] J.E. Burns, J. Pachl, Uniform self-stabilizing rings, ACM Trans. Programming Languages Systems 11 (1989) 330–344.
[7] A.K. Datta, C. Johnen, F. Petit, V. Villain, Self-stabilizing depth-first token circulation in arbitrary rooted networks, Distributed Comput. 13 (4) (2000) 207–218.
[8] E. W. Dijkstra, Self-stabilizing systems in spite of distributed control, In: EWD 391 Selected Writings on Computing: A Personal Perspective, (1973) pp. 41–46.
[9] E.W. Dijkstra, Self stabilizing systems in spite of distributed control, Commun. Assoc. Comput. Mach. 17 (1974) 643–644.
[10] E.W. Dijkstra, Guarded commands and formal derivation of programs, Commun. Assoc. Comput. Mach. 18 (8) (1975) 453–457.
[11] N. Francez, Fairness, Springer Verlag, New York, 1986.
[12] S. Ghosh, A. Gupta, S.V. Pemmaraju, A fault-containing self-stabilizing algorithm for spanning trees, J. Comput. Inform. 2 (1996) 322–338.
[13] S. Ghosh, X. He, Fault-containing self-stabilization using priority scheduling, Inform. Process. Lett. 73 (3–4) (2000) 145–151.
[14] S. Ghosh, S. V. Pemmaraju, Tradeoffs in fault-containing self-stabilization, In: Proceedings of the Third Workshop on Self- Stabilizing Systems, Carleton University Press, 1997, pp.157–169.
[15] M. G. Gouda, F. Haddix, The linear alternator, In: Proceedings of the Third Workshop on Self- Stabilizing Systems, Carleton University Press, 1997, pp. 31–47.
[16] M. G. Gouda, F. Haddix, The alternator, In:*Proceedings of the Third Workshop on Self-Stabilizing Systems (published in association with ICDCS99 The 19th IEEE International Conference on Distributed Computing Systems)*, IEEE Computer Society, Silver Spring, MD, 1999, pp. 48–53.
[17] T. Herman, S. Pemmaraju, Error-detecting codes and fault-containing self-stabilization, Inform. Process. Lett. 73 (1–2) (2000) 41–46.
[18] S.T.HuangN.S. Chen, Self-stabilizing depth-first token circulation on networks, Distributed Comput. 7 (1993) 61–66.
[19] M.H. Karaata, A self-stabilizing algorithm for finding articulation points, Internat. Jo. Found. of Comput. Sci. 1 (1999) 33–46.
[20] M.H. Karaata, Self-stabilizing strong fairness under weak fairness, IEEE Trans. Parallel Distributed Systems 12 (4) (2001) 337–345.
[21] M.H. Karaata, A stabilizing algorithm for finding biconnected components, J. Parallel Distributed Comput. 62 (5) (2002) 982–999.
[23] M.H. Karaata, P. Chaudhuri, A Dynamic Self-Stabilizing Algorithm for Constructing a Transport Net, Computing 68 (2) (2002) 143–161.
[24] S. Katz, K.J. Perry, Self-stabilizing extensions for message-passing systems, Distributed Comput. 7 (1993) 17–26.
[25] M. Mizuno, H. Kakugawa, A timestamp based transformation of self-stabilizing programs for distributed computing environments, in: Proceedings of the 10th International Workshop on Distributed Algorithms (WDAG96)1996, pp. 304–321.

[26] M. Mizuno, M. Nesterenko, A transformation of self-stabilizing serial model programs for asynchronous parallel computing environments, Inform. Process. Lett. 66 (6) (1998) 285–290.
[27] M. Nesterenko, A. Arora, Stabilization-preserving atomicity refinement, in: DISC99 Distributed Computing 13th International Symposium, Springer, Berlin, 1999, 254–268.
[28] F. Petit, V. Villain, A space-efficient and self-stabilizing depth-first token circulation protocol for asynchronous message-passing systems, in: Euro-par' 97 Parallel Processing, Proceedings Lecture Notes in Computer Science, Vol. 1300, Springer, Berlin, 1997, pp.476–479.